Available online at www.sciencedirect.com
SciVerse ScienceDirect
Procedia Computer Science 9 (2012) 802 – 811
Procedia Computer Science
www.elsevier.com/locate/procedia

International Conference on Computational Science, ICCS 2012

An Algorithm for Excluding Redundant Assessments in a Multiattribute Utility Problem

Yerkin G. Abdildin* and Ali E. Abbas

Department of Industrial and Enterprise Systems Engineering, College of Engineering, University of Illinois at Urbana-Champaign, 104 South Mathews ave., Urbana, IL 61801, USA

Abstract

The construction of a multiattribute utility function (MUF) is a fundamental step in decision analysis and can be a difficult task to perform unless some decomposition of the utility function is performed. When partial utility independence conditions exist, the functional form is decomposed into a number of lower-order utility assessments. Often the functional form, resulting from such independence conditions, includes duplicate and redundant assessments. This paper introduces a *twos-complement exclusion algorithm* for determining the minimal set of utility assessments required for a MUF with partial utility independence. The algorithm uses a *ternary matrix representation* of utility assessments. A comparison with a "brute-force" approach is also provided.

Keywords: exclusion algorithm, elimination, deletion, removing duplicate and redundant utility assessments, ternary matrix representation

1. Introduction

There are many situations in life where people make decisions under uncertainty. Decision analysis [1] provides a normative methodology [2] for thinking about the decision problem. In decisions with uncertainty and multiple objectives, a multiattribute utility function, $U(x_1, x_2, ..., x_n)$, is required to represent trade-offs over the different attributes. Examples of attributes in a medical decision-making problem could be health state and wealth. The larger the number of attributes, n, the larger the order of utility assessments needed for the functional form. If there are no independence conditions among the attributes, then the order of utility assessments is equal to n. If the attributes have "mutual utility independence" [3] or even "partial utility independence" conditions [4], then the order of the required utility assessments can be drastically reduced.

Definition 1 [4]: *A set of attributes* X_K *is utility independent (UI) of another set* X_I *given* X_D, *written* $(X_K \ UI \ X_I \mid X_D)$, *with* $X_I \cup X_D = \bar{X}_K$, *if preferences for joint lotteries over* X_K *do not depend on the instantiations of* X_I.

This partial utility independence condition also implies that the normalized conditional utility function

* Corresponding author. Tel.: +1-217-244-8033; fax: +1-217-244-5705.
E-mail address: abdildin@illinois.edu.

doi:10.1016/j.procs.2012.04.086

$U(x_K \mid x_I, x_D) = U(x_K \mid x_I^0, x_D)$, and it also corresponds to the following functional form

$$U(x_K, x_I, x_D) = U(x_K^0, x_I, x_D) + [U(x_K^*, x_I, x_D) - U(x_K^0, x_I, x_D)]U(x_K \mid x_I^0, x_D), \quad (1)$$

where $U(x_K \mid \bar{x}_K)$ is a *normalized conditional utility function* for a set of attributes X_K; x_K^0 and x_K^* are the *least* and *most* preferred values of all the attributes in X_K (respectively), and x_K represents an instantiation of X_K.

To illustrate, consider a situation with four attributes. The condition $(X_3\ UI\ X_4 \mid X_1, X_2)$ implies that $U(x_3 \mid x_1, x_2, x_4) = U(x_3 \mid x_1, x_2, x_4^0)$, and so

$$U(x_1, x_2, x_3, x_4) = U(x_1, x_2, x_3^0, x_4) + [U(x_1, x_2, x_3^*, x_4) - U(x_1, x_2, x_3^0, x_4)]U(x_3 \mid x_1, x_2, x_4^0). \quad (2)$$

This functional form requires three different utility assessments: $U(x_1, x_2, x_3^0, x_4)$, $U(x_1, x_2, x_3^*, x_4)$, and $U(x_3 \mid x_1, x_2, x_4^0)$. For example, $U(x_1, x_2, x_3^0, x_4)$ implies assessments of the attributes X_1, X_2, and X_4 on their entire domains when the attribute X_3 is fixed at instantiation x_3^0. Similarly, $U(x_1, x_2, x_3^*, x_4)$ requires utility assessments of size three when X_3 is fixed at instantiation x_3^*. The normalized conditional utility function, $U(x_3 \mid x_1, x_2, x_4^0)$, needs assessments of X_1, X_2, and X_3 when the attribute X_4 is set arbitrarily to the instantiation, x_4^0. As we can see from this example, when partial UI conditions exist, we can reduce the order of utility assessments of size four, $U(x_1, x_2, x_3, x_4)$, to three assessments of size three. Notice, there are two identical terms $U(x_1, x_2, x_3^0, x_4)$; hence, one of them is a *duplicate* and should be excluded from the list of the required utility assessments.

Recent work in the theory of decision analysis allows the decomposition of the MUF with partial UI conditions. The *iterative decomposition approach* [4] was suggested to decompose the MUF into the lower-order utility assessments. The *multiattribute utility tree* [5] was also introduced to derive the functional form by decomposition into binary gambles. Other decompositions use *fractional hypercubes* [6] and *interpolations* [7, 8] to simplify the construction of the MUF. A full view of these and other methods, which are based on the expected utility of von Neumann and Morgenstern [9], can be found in [10].

This paper proposes a simple exclusion algorithm for determining the set of utility assessments required for the MUF. We first describe utility assessments and illustrate how duplicate and redundant assessments arise in decision analysis. The inputs to the approach are the assessments that follow from the decomposition, and the outputs are the independence assessments that need to be elicited from the decision maker. We propose a ternary matrix representation of utility assessments, which allows us to effectively encode the assessments and operate upon them. We present a matrix-based algorithm for excluding duplicate and redundant utility assessments. We also compare the brute-force approach to our algorithm.

The rest of the paper is organized as follows. In the next Section, we briefly illustrate how utility assessments arise and then propose the ternary matrix representation to encode them. In Section 3, we demonstrate two versions of the exclusion algorithm and provide their asymptotic analyses. The brute-force version is shown only for comparison reasons with our twos-complement method. We present and discuss the simulation results in Section 4 and conclude with the summary and directions for future work in Section 5.

2. Iterative Decomposition and the Ternary Representation

The condition of UI allows us to reduce the order of utility assessments required for the construction of the MUF. Such reduction would significantly minimize the order of assessments needed from the decision maker. The iterative decomposition algorithm [4] addresses this problem by expanding attributes, which assert partial UI conditions, in the form of a tree. As shown in [4], the expansion of the MUF through one attribute, say x, leads to a functional form

$$U(x, \bar{x}) = U(x^*, \bar{x})U(x \mid \bar{x}) + U(x^0, \bar{x})\bar{U}(x \mid \bar{x}), \quad (3)$$

where $\bar{U}(x_K \mid \bar{x}_K) = 1 - U(x_K \mid \bar{x}_K)$ is a *normalized conditional disutility function.* A further expansion through the second attribute, y, results in

$$U(x,y,\overline{xy}) = U(x^*,y^*,\overline{xy})U(x\mid\overline{x})U(y\mid x^*,\overline{xy}) + U(x^*,y^0,\overline{xy})U(x\mid\overline{x})\overline{U}(y\mid x^*,\overline{xy}) + U(x^0,y^*,\overline{xy})\overline{U}(x\mid\overline{x})U(y\mid x^0,\overline{xy}) + U(x^0,y^0,\overline{xy})\overline{U}(x\mid\overline{x})\overline{U}(y\mid x^*,\overline{xy}). \quad (4)$$

By induction, further expansions of the MUF produce the *basic expansion Theorem* [4]:

$$U(x_1,x_2,...,x_n) = \sum_{x_K^{*0}\in X_K^{*0}} U(x_K^{*0},\overline{x}_K)\prod_{i\in K} g(x_i\mid x_{iP}^{*0},x_{iF}), \quad (5)$$

where X_K is the set of attributes that are expanded, $g(\cdot)$ is either a conditional utility or disutility function depending on the instantiations of x_i in the terms $U(x_K^{*0},\overline{x}_K)$; X_{iP} denotes the set of attributes expanded prior to X_i and, X_{iF}, denoting the set of attributes to be expanded after X_i, such that $X_{iP}\cup X_{iF} = \overline{X}_i$. The following example presents a motivation for the proposed algorithm.

Motivating Example. Suppose we have an assertion of the following partial utility independence conditions:

$$(X_3\ UI\ X_4 \mid X_1, X_2) \text{ and } (X_4\ UI\ X_2, X_3 \mid X_1).$$

Using a normalized conditional utility function, we can express these conditions as

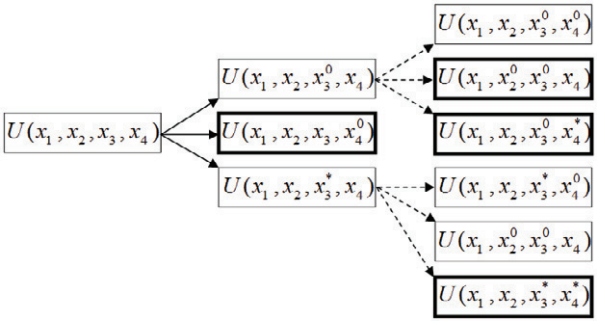
- Attribute X_3 asserts: $U(x_3 \mid x_1, x_2, x_4) = U(x_3 \mid x_1, x_2, x_4^0)$.
- Attribute X_4 asserts: $U(x_4 \mid x_1, x_2, x_3) = U(x_4 \mid x_1, x_2^0, x_3^0)$.

Expanding the MUF through the attributes X_3 and X_4 by (5) and incorporating the given UI assertions into the basic expansion Theorem [4], we have

$$\begin{aligned} U(x_1,x_2,x_3,x_4) &= U(x_1,x_2,x_3^*,x_4^*)U(x_3\mid x_1,x_2,x_4^0)U(x_4\mid x_1,x_2^0,x_3^0) \\ &+ U(x_1,x_2,x_3^*,x_4^0)U(x_3\mid x_1,x_2,x_4^0)\overline{U}(x_4\mid x_1,x_2^0,x_3^0) \\ &+ U(x_1,x_2,x_3^0,x_4^*)\overline{U}(x_3\mid x_1,x_2,x_4^0)U(x_4\mid x_1,x_2^0,x_3^0) \\ &+ U(x_1,x_2,x_3^0,x_4^0)\overline{U}(x_3\mid x_1,x_2,x_4^0)\overline{U}(x_4\mid x_1,x_2^0,x_3^0). \end{aligned} \quad (6)$$

Excluding *duplicate* assessments from (6), gives the terms: $U(x_3\mid x_1,x_2,x_4^0)$, $U(x_4\mid x_1,x_2^0,x_3^0)$, $U(x_1,x_2,x_3^*,x_4^*)$, $U(x_1,x_2,x_3^*,x_4^0)$, $U(x_1,x_2,x_3^0,x_4^*)$, and $U(x_1,x_2,x_3^0,x_4^0)$. We would like to represent these terms using utility assessments and not conditional utility assessments.

A convenient representation of the utility assessments required in (6) is the tree representation shown in Fig. 1(a) that was introduced in [4]. Recall, equation (1) requires the following three utility assessments based on UI assertion of the attribute X_3: $U(x_1,x_2,x_3^0,x_4)$, $U(x_1,x_2,x_3^*,x_4)$, and $U(x_3\mid x_1,x_2,x_4^0)$. These three terms respectively correspond to $U(x_1,x_2,x_3^0,x_4)$, $U(x_1,x_2,x_3^*,x_4)$, $U(x_1,x_2,x_3,x_4^0)$ in the first branch in the tree. Notice that the first two utility assessments are derived from $U(x_1,x_2,x_3,x_4)$ by replacing the expanding attribute X_3 into its least and most preferable instantiations, respectively. The third term, $U(x_1,x_2,x_3,x_4^0)$, incorporates UI conditions asserted by the expanded attribute, X_3. By induction, we expand further these three terms incorporating the UI conditions provided by the attribute X_4. The leaves of the expanded tree contain the list of seven utility assessments as illustrated in Fig. 1(b). As we mentioned, however, there are only six independent assessments for this MUF. How do we exclude the redundant assessments?



2	2	2	0
2	2	0	0
2	0	0	2
2	2	0	1
2	2	1	0
2	0	0	2
2	2	1	1

Fig. 1. (a) Expansion tree for four-attribute problem (b) Utility assessments, and their (c) Ternary matrix representation, **M**

Obviously, the 6th term from the top of the list in Fig. 1(b), namely, $U(x_1,x_2^0,x_3^0,x_4)$, is a *duplicate* of the 3rd one and must be deleted. The closer look on other functions of the list shows that the 2nd and the 5th utility assessments

are *redundant* due to the 1st function, because they are subsets of the 1st one, i.e. they are "covered" by the 1st function. Notice, the only difference between them is in attribute X_3, which is in its "full" state in the 1st utility function, while it is in states x_3^0 and x_3^* in the 2nd and the 5th functions, respectively. Therefore, we should exclude the 2nd, $U(x_1, x_2, x_3^0, x_4^0)$, and the 5th, $U(x_1, x_2, x_3^*, x_4^0)$, utility assessments. There are no other duplicate and redundant functions: the four utility assessments that are required for the MUF are shown in bold rectangles in Fig. 1(a). However, the larger the number of functions and attributes of the decision-making problem, the larger the complexity of the process of checking and eliminating redundancy. This requires an automation of the process. The question is what would be the best representation of the list of functions as in Fig. 1(b), and how the list can be checked efficiently?

In order to automate the process of elimination of duplicate and redundant assessments from the list of functions, we need some reliable and simple representation of utility functions. Such representation should be compact, memory efficient, and intuitive. We found that a *ternary matrix representation* can serve for this purpose.

Definition 2: *The ternary matrix of utility assessments,* ***M****, is an* $m \times n$ *matrix storing integers 0, 1, and 2, representing respectively the least preferable value, the most preferable value, and all values of an attribute, where* m *is the number of utility assessments, and* n *is the number of attributes of the decision-making problem.*

The ternary matrix representation is based on the base three number system, in which ternary digits, or trits (analogous to bits in the binary number system) can be exactly in one of the three states: 0, 1, or 2. A very good analysis of the base-3 number system can be found here [11]. Comparing to the balanced ternary representation [12], which uses -1, 0, and 1 as trits, the ordinary ternary notation ideally matches to our needs. For example, we can represent the instantiation, x^0, the least preferable value of an attribute as 0, x^*, the most preferable value as 1, and, x, which represents all values, as 2. By definition, the following two conditions must hold for any instantiation of an attribute: (i) 2s include both 1s and 0s, and (ii) 1s and 0s are distinct.

We can now encode any utility assessment into a sequence of zeros, ones, and twos, or a *ternary vector* [13]. For example, the first utility function in Fig. 1(b), $U(x_1, x_2, x_3, x_4^0)$, will be represented by the ternary vector: 2 2 2 0. Similarly, we can encode other six functions and create a 7×4 ternary matrix **M** as shown in Fig. 1(c). The ternary matrix representation significantly simplifies the comparison of elements of **M** since 2 > 1, 2 > 0, and $1 \neq 0$. Some other applications of the ternary vector we have found in the literature include a three-valued logic, where the logical values "false," "true," and "maybe," can be denoted respectively by 0, 1, and * [13].

In general, given a list of m functions of n attributes, we will represent the list as $m \times n$ ternary matrix. Although there might be representations other than the matrix form, the ternary matrix representation gives us the flexibility in developing algorithms and connecting them to other methods in decision analysis. We now ready to present the exclusion algorithm.

3. The Exclusion Algorithm

Having encoded the given list of utility functions into the ternary matrix **M** as shown in Fig. 1(c), we can easily exclude the possible duplicate and redundant utility assessments as following. First, we develop an algorithm that eliminates the duplicate and redundant rows from matrix **M**, dynamically adjusting its size in the process. Then, we decode the resulting output matrix **M** back into the list of utility assessments. In this Section, we present two versions of the exclusion algorithm: (i) the brute-force approach, and (ii) the twos-complement approach.

3.1. Exclusion Algorithm Version 1: Brute-Force Approach

The brute-force exclusion algorithm is straightforward; it compares each row of the ternary matrix **M** with all other rows element-wise and excludes duplicate and redundant rows as they are found. If all elements of the 1st (top) row of **M** are greater than or equal to the corresponding elements of the 2nd row, then the 2nd row can be deleted from **M** as either duplicate or redundant, namely:

(i) if both rows are identical, then row two is duplicate;

(ii) if all elements of the 1st row, which are greater than the corresponding elements of the 2nd row, are twos, then the 2nd row is redundant.

Fig. 2(a) illustrates the comparison of the first two rows of matrix **M** from Fig. 1(c), where row two is redundant.

Row 1	2	2	**2**	0	Row 3	2	0	0	2	Row 5	2	2	1	0
Row 2	~~2~~	~~2~~	~~0~~	~~0~~	Row 6	~~2~~	~~0~~	~~0~~	~~2~~	Row 7	2	2	1	**1**

Fig. 2. (a) Row 2 of matrix **M** from Fig. 1(c) is redundant due to row 1, because all elements of row 1 ≥ elements of row 2, and the element (in dark red), which is > than the respective element of row 1, is equal to two; (b) Row 6 is a duplicate of row 3; hence, deleted; (c) Row 7 dominates row 5, but cannot exclude it, because the 4th element of row 7, which is greater than the corresponding element of row 5, is equal to one.

Fig. 2(b) depicts the exclusion of row six that is a duplicate of row three. Notice, however, if we compare rows five and seven as shown in Fig. 2(c), the former does not dominate the latter, but the latter dominates the top row; hence, the order is important. Also notice, even that row seven dominates row five element-wise, the latter cannot be eliminated from the matrix, because the element of the dominating row that is greater than the corresponding element of the dominated row is equal to one.

Table 1 presents the high-level pseudocode of the brute-force algorithm, `exclusionV1`. The algorithm contains two nested loops, an outer loop by index r and an inner loop by index i. Starting with the first top row of the ternary matrix **M**, it goes down row by row through the matrix (*outer loop's row*) comparing every other row (*inner loop's row*) with the current outer row r, and eliminating the inner row i, if necessary. Notice, in line 6, if any element of row r is less than the respective element of i, we skip row i. The rest of the pseudocode in Table 1 is self-explaining (please see the detailed pseudocode in the Appendix).

Table 1. High-level pseudocode of algorithm exclusionV1

Algorithm exclusionV1(**M**):
Input: An $m \times n$ ternary matrix **M** storing integers: 0, 1, 2.
Output: The matrix **M** without duplicate and redundant rows.

```
1:  Set index r to be the 1st (unprocessed) row of the matrix M.
2:  WHILE an unprocessed row of M exists DO compare all other rows of M with row r as follows:
3:      Set index i to be the 1st (uncompared) row of the matrix M.
4:      WHILE an uncompared row of M exists DO following:
5:          IF i ≠ r THEN
6:              FORALL n: compare rows r and i element-wise and break out of the For loop IF any element of r < i
7:              ELSEIF all elements of r are greater than or equal to the corresponding elements of row i THEN
8:                  IF all of them are equal THEN delete row i since it is a duplicate of row r
9:                  ELSE
10:                     FORALL n:
11:                         IF all elements of row r, which are greater than the corresponding elements of row i, are equal to 2 THEN
12:                             delete row i since it is redundant due to row r
13:                         ESLEIF any element of row r, which is greater than the corresponding element of row i, is equal to 1 THEN
14:                             break out of the For loop
15: RETURN M
```

We now provide asymptotic analysis [14, 15] for the running time of algorithm `exclusionV1`. The *running time* is measured by counting the number of primitive operations [16, 17] that are executed on the best and worst inputs of the algorithm. An example of the primitive operation is assigning a value to a variable, comparing two numbers, or performing an arithmetic operation as shown in pseudocodes in the Appendix. The best input of the exclusion algorithm is considered to be a ternary matrix with only 2s in its first top row. In the worst case, the input matrix will not contain any duplicate and redundant rows; all rows of matrix **M** can be unique. Proof of Proposition 1 is omitted due to space limitations.

Proposition 1. *The growth rate for the running time of algorithm* `exclusionV1` *for excluding duplicate and redundant assessments (rows) from an* $m \times n$ *ternary matrix of utility assessments is* $\Theta(nm)$ *in the best case and is* $\Theta(nm^2)$ *in the worst case.*

3.2. Exclusion Algorithm Version 2: Twos-Complement Method

Similar to the brute-force algorithm, the *twos-complement exclusion algorithm* eliminates duplicate and redundant rows of the ternary matrix **M** by comparing each row of the matrix with all other rows. Unlike the brute-force approach, however, the twos-complement method compares the rows of **M** partly. In addition, if a full row of

twos is found in the matrix, the twos-complement algorithm immediately terminates returning **M** with a single row of twos.

The main idea of the twos-complement approach is the following. Since twos in **M** include ones and zeros by definition, when comparing row i with row r element-wise, the twos-complement algorithm makes the comparison only by the elements, whose values in row r are the *complements of twos*, i.e. ones and zeros. Hence, starting with the 1st (top) row, search for elements that are complements of 2, i.e. 0 or 1. Identify their corresponding columns. Any other row that has identical elements in the corresponding columns is duplicate or redundant. Fig. 3(a, b) demonstrates exclusion of rows two and five from matrix **M** due to their redundancy with row one. Fig. 3(c) shows the elimination of row six as a duplicate of row three.

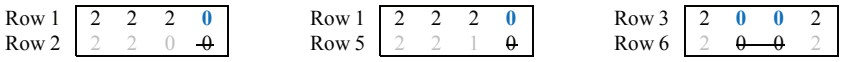


Fig. 3. (a) Row 2 of matrix **M** from Fig. 1(c) is redundant due to row 1, because the element of row 2 in the 4th column is equal to the element (in blue) in the twos-complement column of row 1; hence, eliminated; (b) Row 5 is redundant due to row 1; therefore, deleted; (c) Row 6 duplicates row 3 and will be excluded, because its elements in columns 2 and 3 are identical to the elements of the twos-complement columns of row 3.

The algorithm first stores the indices of twos-complement elements of row r in a vector, `twos_compl`.

Definition 3: *The indices of columns of the ternary matrix **M**, which do not contain 2s in the current row r being processed, create a vector of twos' complements,* `twos_compl`*, for row r.*

Then, the algorithm compares only parts of the rows r and i, by the columns, which indices are found in the `twos_compl` vector of row r, and excludes row i, if necessary. Hence, the name of the algorithm is the "twos-complement exclusion algorithm."

Table 2 presents the high-level pseudocode of the twos-complement exclusion algorithm `exclusionV2` (the more detailed pseudocode is in the Appendix). The algorithm creates vector `twos_compl` in line 4 to store indices of 0s and 1s of the current outer row r. In line 5 it returns a vector of twos and terminates the program if `twos_compl` is empty; otherwise, it compares elements of rows r and i by the indices in `twos_compl` and, if all of them are pairwise identical, removes the row i. The twos-complement exclusion algorithm minimizes the comparison of all elements of the rows of matrix **M**, and it is even more efficient when attributes are mostly utility dependent (more twos in the rows of the ternary matrix). The extra space requirement of `exclusionV2` due to vector `twos_compl` is in $O(n)$ in the worst case.

Table 2. High-level pseudocode of algorithm exclusionV2

	Algorithm exclusionV2(**M**):
	Input: An *m* x *n* ternary matrix **M** storing integers: 0, 1, 2.
	Output: The matrix **M** without duplicate and redundant rows.
1:	Initialize a vector *twos_compl* to store indices of 1s and 0s of row *r*.
2:	Set index *r* to be the 1st (unprocessed) row of matrix ***M***.
3:	**WHILE** an unprocessed row of ***M*** exists **DO** compare all other rows of ***M*** with row *r* as follows:
4:	**FORALL** *n:* **IF** an element of row *r* is not equal to two **THEN** set its (column) index into vector *twos_compl.*
5:	**IF** *twos_compl* is empty **THEN** delete matrix **M,** return a vector of *n* twos, and terminate the program.
6:	Set index *i* to be the 1st (uncompared) row of matrix ***M.***
7:	**WHILE** an uncompared row of ***M*** exists **DO** following:
8:	**IF** $i \neq r$ **THEN**
9:	**IF** any corresponding elements of rows *r* and *i* indexed in *twos_compl* are different **THEN** take next *i.*
10:	**ELSEIF** all of them are equal **THEN** delete row *i* since it is either *redundant* or *duplicate*
11:	**RETURN** ***M***

Proposition 2. *The twos-complement exclusion algorithm* `exclusionV2` *runs in* $\Theta(n)$ *in the best case and in* $\Theta(nm^2)$ *in the worst case.*

Sketch of the Proof:

Correctness: The input $m \times n$ ternary matrix **M** either contains a vector of 2s or does not contain it. In the former case, `exclusionV2` returns the vector of 2s and terminates (line 5). In the latter case, all possible duplicate and redundant rows will be removed from **M** in line 10, and the algorithm terminates in line 11 returning the updated

matrix **M**. Hence, the algorithm works correctly for any input.

Running time: The best input of `exclusionV2` should contain the vector of 2s in the first (top) row of matrix **M**; that is, `twos_compl` of the 1st row should be empty. In line 4, the algorithm first creates the vector `twos_compl` for the current row r, which requires n comparison operations. Then it checks the vector `twos_compl` for emptiness and, since `twos_compl` is empty, deletes matrix **M**, returns the vector of 2s, and terminates (line 5). Therefore, `exclusionV2` runs in $\Theta(n)$ in the best case.

In the worst case, the input matrix will not contain the vector of 2s at all; all rows of matrix **M** can be unique. The algorithm `exclusionV2` has two nested loops in lines 3 and in line 7, each repeated $m+1$ times at most; and their bodies in lines 4-10 and 8-10, respectively, each repeated m times at most. This would require $O(m^2)$ row comparisons for the number of elements indicated by the size of vector `twos_compl`, which is in $O(n)$ in the worst case; therefore, the growth rate for the running time of the twos-complement algorithm `exclusionV2` is in the set $O(nm^2)$, in the worst case. Analogously, the worst-case time complexity of `exclusionV2` is in the set $\Omega(nm^2)$. Since the growth rate is in $O(nm^2)$ and it is in $\Omega(nm^2)$, it is $\Theta(nm^2)$, in the worst case. Q.E.D.

Although, both algorithms have, similar up to constant factors, asymptotic upper and lower bounds in the worst case, the twos-complement algorithm outperforms the brute-force algorithm in simulations, as shown in the next section. In addition, twos-complement algorithm asymptotically outperforms the brute-force algorithm for the best-case input. Moreover, the larger the number of twos in rows of the input ternary matrix **M**, the smaller the number of operations that the twos-complement algorithm demands compared to the brute-force algorithm. This means that the twos-complement algorithm works faster especially when the attributes of the decision-making problem provide a milder set of UI conditions, i.e. the problem has higher complexity. We now provide simulation analyses of the brute-force and twos-complement algorithms.

4. Simulation Results

We implemented both versions of the algorithm in C++ with Microsoft Visual C++ 2010 Express on a PC with the following environment: Intel® Core™ i5-2320, 6GB SDRAM, Windows® 7 Home Premium, 64Bit. The input matrix **M** was implemented as a "vector of vectors:" the sequential representation of functions can be more space efficient than the linked representations [18]. For example, representing a utility assessment $U(x_1, x_2, ..., x_n)$ as a vector of short integers requires $2n$ bytes, while one-way linked list representation would require $6n$ bytes, i.e., 4 bytes for each pointer and 2 bytes for each item of the function. The memory efficiency of the implementation is very important for certain applications.

We then ran simulations in the following order:

(i) Randomly generate the ternary matrix **M** with the given number of rows *m* and columns *n*.

(ii) Run brute-force `exclusionV1` with **M**; keep its output **M1** and accumulate its running time in *Time1*.

(iii) Run twos-complement `exclusionV2` with **M**; keep its output **M2**, accumulate its running time in *Time2*.

(iv) If **M1**≠**M2**, then output: "Error!" and terminate the simulation.

(v) Delete matrices **M**, **M1**, **M2**.

(vi) Repeat steps from (i) to (v) 10,000 times.

(vi) Return average running times of 10,000 tests as *AverageTime1* and *AverageTime2*.

We compared the algorithms in two types of tests. In the first type, we changed the number of functions, m, keeping the number of attributes, n, fixed. In the second, we varied n keeping m fixed. We randomly generated $m \times n$ ternary matrix **M** synchronizing the seed of the random generator with the time function as follows: srand(time(0) + *i* * 2 + 1), where *i* is the current test number, which was incremented from 0 to 9,999. This way the random generator provided completely different matrices, which served as input for both versions of the algorithm. The time of matrix generation was not included in the running times of the algorithms.

Fig. 4 illustrates average running times of 10,000 tests when (a) the number of functions varies from 50 to 250 with step 50 for a seven-attribute decision problem, (b) the number of attributes increases from 3 to 11 for a fixed number of functions, 150. We see that the twos-complement algorithm shows much better performance. Fig. 5 demonstrates average running times of the algorithms for different input matrices in 2D, and Fig. 6 in 3D. Again, the twos-complement algorithm shows better running times on the whole domain of m and n.

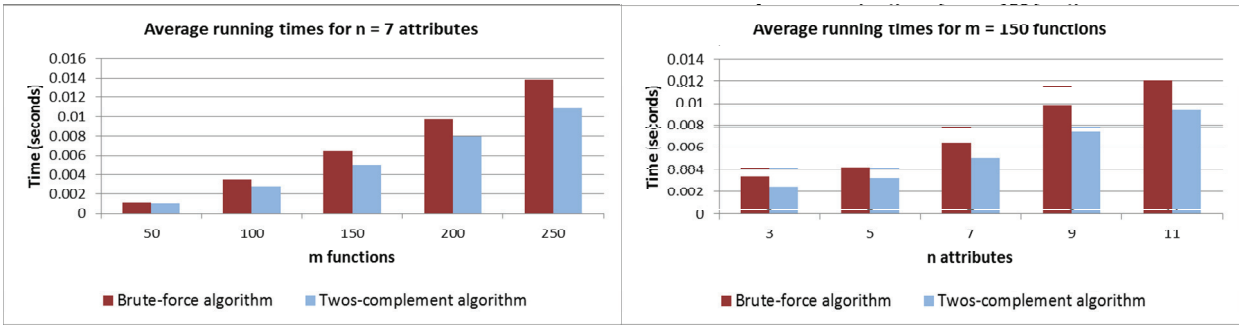


Fig. 4. Average running times for: (a) fixed number of attributes; (b) fixed number of utility functions.

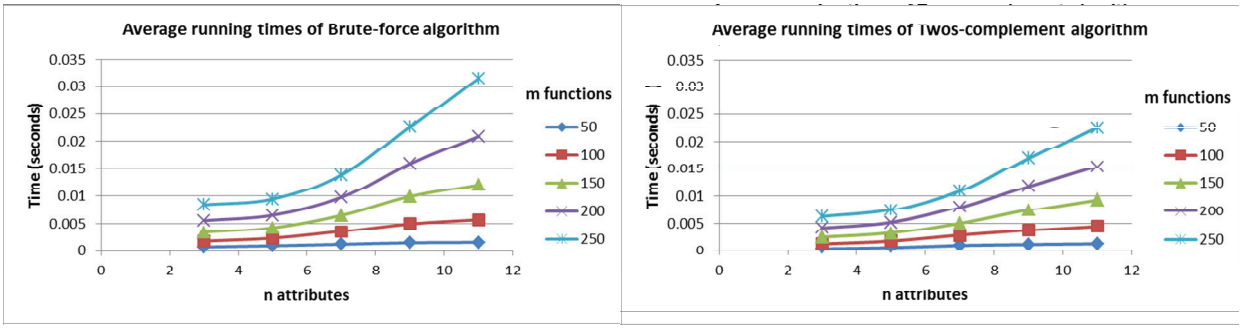


Fig. 5. Average running times for different number of attributes and functions for: (a) Brute-force algorithm; (b) Twos-complement algorithm

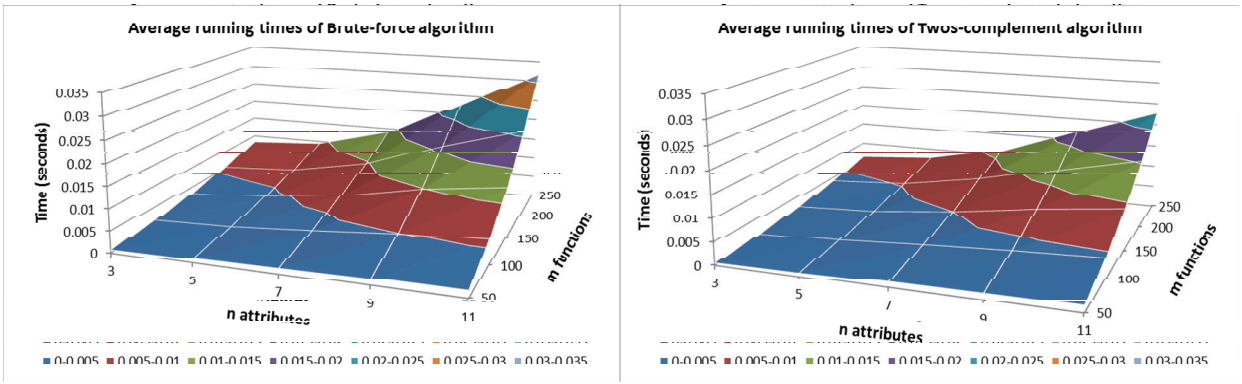


Fig. 6. Average running times in 3D for: (a) Brute-force algorithm; (b) Twos-complement algorithm

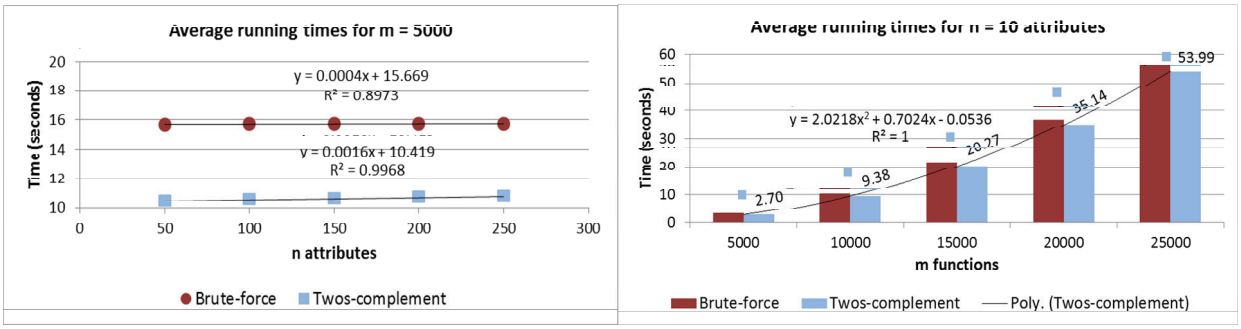


Fig. 7. Average running times of the algorithms for larger problems with: (a) different number of attributes; (b) different number of functions

We also compared algorithms for larger sets of problems. Fig. 7 illustrates average running times of the algorithms when (a) n changes from 50 to 250 while m is fixed at 5,000; and (b) m varies from 5,000 to 25,000 while n is fixed at 10. As we can see, both algorithms are linear in the number of attributes and quadratic in the number of utility assessments. These results are consistent with our asymptotic analyses. In summary, the algorithms demonstrate efficient running times with considerably better performance of the twos-complement algorithm.

5. Conclusion and Future work

Elimination of duplicate and redundant functions is an important problem in the field of decision analysis. We presented a method for excluding duplicate and redundant utility assessments in a multiattribute decision-making problem. We suggested using the base-3 number system to represent the given list of utility assessments in the ternary matrix format. The base-3 number system ideally encodes utility functions used in decision analysis. We introduced two versions of the exclusion algorithm; the brute-force version was implemented only for comparison purposes with our twos-complement exclusion algorithm. The asymptotic analysis of our algorithm proves its efficiency.

Our simulation results demonstrate excellent running time performance of the twos-complement exclusion algorithm for a random input of an arbitrary size when compared with the brute force approach. However, there are still many challenges and room for future work. The exponential complexity of decision-making problems demands the fastest algorithms. Although we have developed an efficient exclusion algorithm, there might be some other approaches to address the problem. For example, one can consider using hash tables or other techniques to improve the running time of the proposed exclusion algorithm. Future work may also include the implementation aspects or a particular application of the exclusion algorithm.

Acknowledgements

This work is supported in part by the National Science Foundation CAREER award NSF-DRMS 0846417. We would like to thank the anonymous reviewers for their helpful comments.

References

1. R.A. Howard, Decision Analysis: Applied Decision Theory. Proceedings of the 4th Int-l Conf. on Operational Research (1966) 55–77.
2. R.A. Howard, In praise of the old time religion. Utility Theories: Measurements and Applications. Kluwer, Boston, 1992.
3. R.L. Keeney and H. Raiffa, Decisions with Multiple Objectives: Preferences and Value Tradeoffs. Wiley, New York, 1976.
4. A.E. Abbas, General Decompositions of Multiattribute Utility Functions with Partial Utility Independence. J. MCDA. 17 (2010) 37–59.
5. A.E. Abbas, The Multiattribute Utility Tree. Decision Analysis 8 (2011) 165–169.
6. P.H. Farquhar, A fractional hypercube decomposition theorem for multiattribute utility functions. Operations Research 23 (1975) 941–967.
7. D.E. Bell, Multiattribute utility functions: Decompositions using interpolation. Management Science 25 (1979) 744–753.
8. D.E. Bell, Consistent assessment procedures using conditional utility functions. Operations Research 27 (1979) 1054–1066.
9. J. von Neumann and O. Morgenstern, Theory of Games and Economic Behavior. Princeton University Press, Princeton, NJ, 1947.
10. A.E. Abbas, Constructing Multiattribute Utility Functions for Decision Analysis. TutORials in Oper-s Research 7 (2010) 62–98.
11. B. Hayes, Third Base. American Scientist v.89, No. 6 (2001)
12. D. E. Knuth, The Art of Computer Programming, Volume 2: Seminumerical Algorithms. Addison-Wesley, p.207, reprint of 2000.
13. D. E. Knuth, The Art of Computer Programming, Volume 4A: Combinatorial Algorithms. Pearson Education, Inc., p.163, reprint of 2011.
14. C.A. Shaffer, (2nd eds.) A Practical Introduction to Data Structures and Algorithm Analysis. Prentice Hall, pp.49-79, 2001.
15. T.H. Cormen, C.E. Leiserson, and R.L. Rivest. Introduction to Algorithms. The MIT Press, pp.1-28, 1997.
16. M.T. Goodrich, R. Tamassia, and D. Mount. Data Structures and Algorithms in C++. John Wiley & Sons, Inc., pp.107-129, 2004.
17. J. Kleinberg and E. Tardos, Algorithm Design. Addison Wesley, pp.29-38, 2006.
18. T.A. Standish, Data Structures, Algorithms, and Software Principles in C. Addison Wesley, pp.301-2, 1995.

Appendix

Table 3. The detailed pseudocodes of the algorithms: (a) Brute-force; (b) Twos-complement.

Algorithm exclusionV1(**M**):
Input: An *m* x *n* ternary matrix **M** storing integers: 0, 1, 2.
Output: The matrix **M** without duplicate and redundant rows.

```
1:  rows ← m
2:  r ← 1                                   // index of outer loop's row
3:  while r < rows + 1 do
4:      i ← 1                               // index of inner loop's row
5:      while i < rows + 1 do
6:          if i ≠ r then
7:              redundant ← false           // flag "redundant"
8:              flgE ← true                 // flag "Equal"
9:              flgGE ← true                // flag "Greater or Equal"
10:             for j = 1 to n do
11:                 if M[r][j] < M[i][j] then
12:                     flgGE ← false
13:                     break               // break out of the for loop
14:             if flgGE = true then
15:                 for j = 1 to n do
16:                     if M[r][j] ≠ M[i][j] then
17:                         flgE ← false
18:                         break           // break out of the for loop
19:                 if flgE = true then
20:                     redundant ← true
21:                 else          // flgGE = true and flgE = false
22:                     redundant ← true
23:                     for j = 1 to n do
24:                         if M[r][j] > M[i][j] & M[r][j] = 1 then
25:                             redundant ← false
26:                             break  // row i is not redundant
27:                 if redundant = true then
28:                     M.erase(M.begin() + i) // delete row i
29:                     rows ← rows - 1
30:                     if i < r then   // if row i was before row r
31:                         r ← r - 1   // shift up rows r and i
32:                     i ← i - 1
33:         i ← i + 1
34:     r ← r + 1
35: return M
```

Algorithm exclusionV2(**M**):
Input: An *m* x *n* ternary matrix **M** storing integers: 0, 1, 2.
Output: The matrix **M** without duplicate and redundant rows.

```
1:  rows ← m
2:  vector twos_compl       // store indices of 0s and 1s of row r
3:  r ← 1                   // index of outer loop's row
4:  while r < rows + 1 do
5:      for j ← 1 to n do
6:          if M[r][j] ≠ 2 then
7:              twos_compl.push_back(j)
8:      if twos_compl.empty() then
9:          vector twos
10:         twos.assign(n,2)
11:         M.clear()
12:         M.push_back(twos)    // return vector of 2s
13:         break                // break out of the WHILE loop
14:     i ← 1                    // inner loop index
15:     while i < rows + 1 do
16:         if i ≠ r then
17:             size ← twos_compl.size()
18:             indentical ← true          // flag "identical"
19:             for j ← 1 to size do
20:                 curr ← twos_compl[j]
21:                 if M[r][curr] ≠ M[i][curr] then
22:                     indentical ← false
23:                     break       // break out of the FOR loop
24:             if indentical = true then
25:                 M.erase(M.begin() + i) // delete row i
26:                 rows ← rows - 1
27:                 if i < r then   // if row i was before row r
28:                     r ← r - 1   // shift up rows r and i
29:                 i ← i - 1
30:         i ← i + 1
31:     twos_compl.clear()          // clear twos_compl of r
32:     r ← r + 1
33: return M
```